C++

Samuele Amato

Contents

1 Storia di C++

I linguaggi di programmazione hanno subbito una grande evoluzione sin dal primo computer elettronico, all'inizio i programmatori lavoravano con quelle che erano le istruzioni più primitive: il **linguaggio macchina**, queste istruzioni erano formate da stringhe di 0 e 1.

Assembly presto diventa lo standard nel mondo della programmazione, rimpiazza (mappa) le ingombranti stringhe binarie con comandi leggibili dall'essere umano come **ADD** e **MOV**.

Essendo che i compiti eseguiti dai software sviluppati divennero sempre più complessi, i programmatori sentirono la necessità di un linguaggio capace di eseguire istruzioni matematiche relativamente complesse, che a loro volta erano combinazione di codici assembly, così nacque **FORTAN**, il primo linguaggio ad **alto livello** ottimizzato per calcoli numerici e scientifici, oltre ad introdurre concetti come sub-routine, funzioni e loop al mondo della programmazione.

Col tempo si sono sviluppati lingiaggi di livello superiore, come **COBOL** e **BASIC**, che hanno consentito ai programmatori di lavorare con qualcosa che si avvicina all'inglese, ad esempio `let i = 6`.

C stesso nasce come evoluzione rispetto alle sue versioni precedenti chiamate **B**, che era una versione migliorata di **BPCL**. Nonostante C fosse nato per aiutare i programmatori ad usare le funzionalità del nuovo hardware di quei tempi è diventato famoso grazie alla suaa velocità e portabilità. **C** è un linguaggio procedurale ed essendo che i computer si sono all'approccio orientato ad oggetti Bjarne Stroustrup inventò **C++**, che continua ad essere uno dei linguaggi di programmazione più utilizzati.

C++ ha implementato concetti della programmazione **object oriented** come l'**ereditarietà, incapsulamento, astrazione, polimorfismo**, termini che saranno spiegati più avanti, C++ continua ad essere utilizzato in molte applicazioni non solo perchè i nuovi linguaggi non soddisfano i requisiti di molte applicazioni, ma anchè per la sua flessibilità e potenza piazzata nelle mani del programmatore. C++ è regolarizzato dallo standard **ANSI** e continua ad evolversi.

2 Interpretatore e Compilatore

Un *interpretatore* traduce ed esegue un programma mentre lo legge, trasformando le istruzioni del programma o il codice sorgente, direttamente in azioni. Un *compilatore* traduce il codice sorgente in una forma intermediaria. Questa fase è chiamata **compilazione** e produce un file oggetto. Un applicazione di collegamento chiamata linker viene eseguita dopo il compilatore e combina il

file oggetto in un programma eseguibile contentente codice macchina che può essere eseguito direttamente sul processore.

Siccome gli interpreti leggono il codice così com'è scritto (leggono e eseguono il codice sorgente nel suo formato originale, senza la necessità di una fase di compilazione separata) ed eseguono immediatamente il codice, questi possono essere più facili da utilizzare per i programmatori. La maggior parte dei programmi interpretati è indicata come *script*, e l'interprete è indicato come *script engine*.

Il **compilatore** introduce lo step extra di compilare il codice sorgente (che è leggibile dagi umani in object code (che è leggibile dalle macchine). Questo step extra potrebbe sembrare non conveniente ma i programmi compilati vengono eseguiti molto velocemente essendo che la task di tradurre il codice sorgente in linguaggio macchina è già stata fatta, e non è richiesta quando viene eseuguito il programma.

Un altro vantaggio dei linguaggi compilati come C++ è che si può distribuire l'eseguibile del programma a chi non ha il compilatore, mentre con un linguaggio interpretato c'è bisogno di avere l'interprete insallato per eseguire il programma su qualsiasi computer.

3 Ciao Mondo!

Anche un programma semplice come quello che segue ha parti interessanti da analizzare.

```
#include <iostream>

int main() {
    std::cout << "Ciao Mondo!\n";
    return 0;
}
```

Analisi del codice

Nella prima linea del codice il file **iostream** viene importato all'interno del file corrente. Ecco come funziona: il primo carattere è il simbolo # che è un segnale per un programma chiamato **pre-processor**, ogni volta che si avvia il compilatore, viene eseguito prima il **preprocessor**, questo legge il codice sorgente cercando per linee che iniziano col simbolo # e agendo su queste.

Il comando **#include** è un istruzione per il preprocessore che dice *ciò che segue è un nome di un file, trova questo file e importalo*, le parentesi angolate intorno al nome del file comunicano al preprocessore di cercare in tutti i luoghi usuali per questo file. Il file **iostream** è utilizzato da **cout** che assiste nella

scrittura sulla console.

L'effetto della riga 1 è quello di includere il file iostream in questo programma come se lo avessi digitato tu stesso.

Funzione main

Il programma inizia con una funzione chiamata **main**, ogni programma scritto in C++ ha una funzione main. Una **funzione** è un blocco di codice che segue una o più azioni. Le funzioni in genere devone essere chiamate per essere eseguite, ma **main** è una funzione speciale, è l'**entry point del programma**, quando il programma viene eseguito la funzione main viene chiamata automaticamente

La funzione main come tutte le funzioni deve specificare il tipo di valore che restituisce, in questo caso il tipo del valore restituito è **int**, il che significa che la funzione restituisce 0 al sistema operativo quando viene eseguita, per convenzione il valore 0 viene usato per indicare che l'esecuzione è andata a buon fine. Questo può essere importante se l'applicazione viene lanciata da altre applciazioni perchè col codice di uscita (exit code) si può comunicare all'applicazione se è stata eseguita o meno con successo.

Tutte le funzioni iniziano con una parentesi graffa aperta e finisce con una parentesi graffa chiusa, tutto ciò che è incluso fra il set di parentesi graffe è parte della funzione.

std::cout

Il cuore di questo programma risiede nell'uso di **std::cout**, l'oggetto cout viene utilizzato per stampare testo sullo schermo (più avanti verranno trattati gli oggetti e dettagliatamente cout e cin), cout è un oggetto fornito dalla libreria standard. Una **libreria** è una collezione di classi.

Nel codice si è fatto capire al compilatore che l'oggetto cout è parte della libreria standard usando l'*identificatore di namespace* **std** essendo che potrebbero esserci più oggetti con lo stesso nome, in questo caso viene comunicato al compilatore di usare **cout** presente all'interno del namespace standard, lo si ha comunicato al compilatore usando **std::** prima di cout.

Ecco come viene utilizzato cout: scrivi la parola cout, seguita dall'operatore di redirezione dell'output (<<). Qualsiasi cosa segua l'operatore di redirezione dell'output viene scritta sulla console. Se desideri scrivere una stringa di caratteri.

4 Usare il Namespace Standard

Dopo un pò usare `std::` prima di ogni cout e altri oggetti della libreira standard risulta fastidioso, per rimediare a questo possiamo usare `using std::cout` oppure `using namespace std`, nel primo caso diciamo che useremo solo `std::cout` quindi basterà scrivere solo `cout`, stessa cosa vale per il secondo caso con l'eccezione che vengono utilizzati tutti gli oggetti.

5 Commenti

Un commento è essenzialmente una parte di testo ignorata dal compilatore che serve a rendere il codice più chiaro. Esistono due tipi di commenti:

```
#include <iostream>

/*
commenti multi-linea
*/

int main() {
    std::cout << "test"; // stampa testo
}
```

6 Funzioni - Introduzione

La funzione `main` è una funzione inusuale, una funzione per essere utile ha bisogno di essere invocata nel corso del programma, durante l'esecuzione del programma la funzione main viene invocata dal sistema operativo.

Un programma è eseguito linea dopo linea nell'ordine in cui vengono scritte finchè una funzione non viene invocata, in quel caso il programma si biforca per eseguire la funzione. Quando la funzione termina restituisce il controllo alla linea di codice successiva alla chiamata della funzione.

```
#include <iostream>


void hello_world() {
    std::cout << "Hello World\n";
}

int main() {
    hello_world();
}
```

In questo caso la funzione hello_world() è di tipo `void`, per questo non ritorna nulla.

Usare una funzione

Una funzione o restituisce un valore o nulla, una funzione che deve eseguire la somma di due numeri dovrà restutire un numero, quindi sarà di tipo **int**, una funzione che stampa del testo su terminale non deve restituire nulla e quindi sarà di tipo **void**. Una funzione consiste in un **header** ed un **body**.

header

L'header è formata da **tipo di ritorno**, **nome funzione**, **parametri funzione**. I **parametri** di una funzione consentono di passare valori alla funzione, ad esempio se una funzione deve sommare due numeri, i numeri sarebbero parametri della funzione.

```
int Somma(int num1, int num2) // esempio di header
```

questo è un esempio di header con due parametri che restituisce un tipo intero.

Un parametro è una dichiarazione di quale tipo di valore verrà passato, il valore effettivamente passato quando la funzione viene chiamata è chiamato argomento.

body

Il body di una funzione è formato da un set di parentesi graffe che al loro interno contengono zero o più istruzioni, le istruzioni costituiscono il funzionamento della funzione.

7 Variabili

In C++ una variabile è un posto dove conservare informazioni, è un luogo nella memoria del computer dove possiamo conservare un valore e prelevarlo.

Visualizzare la memoria del computer come una serie di scomparti può aiutare a capire come vengono gestite le variabili. Ogni scomparto è numerato sequenzialmente ed ha un indirizzo di memoria univoco. Quando si dichiara una variabile sis ta riservando uno o più di questi scomparti per immagazzinare un valore.

Se si immagina ogni scomparto come una piccola cassaforte numerata il nome della variabile è come l'etichetta su una di queste casseforti, il che ti permettie di trovarla facilemnte senza dover consocere il suo effettivo inidirizzo di memoria. Quindi se `var1` inizia all'indirizzo di memoria 103 significa che i dati associati alla variabile sono conservati in quella posizione di memoria e nelle posizioni di memoria successive a seconda delle dimensioni della variabile.

Quando si definsice una variabile in C++ è necessario specificare al compilatore il tipo della variabile, questa informazione serve al compilatore per capire quanto spazio riservare e quale tipo di valore si desidera memorizzare nella variabile, consente anche al compilatore di avvisare in caso si stesse tentando di memorizzare un valore del tipo sbagliato nella variabile, questa è una caratteristica chiamata **strong typing**

Ogni scomparto è di dimensioni pari ad un byte, se il tipo di variabile che si crea è pari a 4 byte, occorrono 4 byte di memoria (quattro scomparti), il tipo di variabile comunica al 1compilatore quantii byte di memoria riservare per la variabile.

8 Grandezza dei numeri interi

Su qualsiasi computer ogni tipo di variabile occupa una quantità di spazio singola e invariabile, un numero intero potrebbe occupare 2 byte su un computer e 4 su un altro, ma su entrambi i computer la quantità occupata rimane costante.

Per memoriazzare numeri interi più piccoli è possibile creare una variabile utilizzando il tipo short, ovvero un intero breve che di solito occupa 2 byte sulla maggior parte dei computer, mentre un intero lungo, **long**, è di solito lungo 4 byte, e un intero senza la keyword long o short può essere di 2 o 4 byte.

Signed e Unsigned

Tutti i tipi interi si presentano in due varianti: **signed** e **unsigned**, qualsiasi numeri intero senza la parola `unsigned` è considerato con segno quindi può essere sia positivo che negativo.

Questi vengono memorizzati nella stessa quantità di spazio e a causa di questo per un numero intero con segno parte dello spazio viene utilizzata per memorizzare dati in caso il numero fosse negativo, ad esempio un intero breve (mem-

orizzato in due byte):

$$signed = -32.000 \quad ; \quad 32.000 \quad mentre \quad unsigned = 0 \quad ; \quad 64.000$$

9 Tipi Fondamentali

Diversi tipi di variabili sono integrati in C++ (fondamentali), questi sono int, float e char. Le variabili float rappresentaano i numeri reali, le variabili char occupano un byte e sono utilizzate per memorizzare i 256 caratteri e simboli dei caratteri ASCII.

Tipo	**Dimensione**	**Valori**
bool	1 byte	Vero o falso
unsigned short int	2 byte	0 a 65.535
short int	2 byte	-32.768 a 32.767
unsigned long int	4 byte	0 a 4.294.967.295
long int	4 byte	-2.147.483.648 a 2.147.483.647
int (16 bit)	2 byte	-32.768 a 32.767
int (32 bit)	4 byte	-2.147.483.648 a 2.147.483.647
unsigned int (16 bit)	2 byte	0 a 65.535
unsigned int (32 bit)	4 byte	0 a 4.294.967.295
char	1 byte	256 valori carattere
float	4 byte	1.2e–38 a 3.4e38
double	8 byte	2.2e–308 a 1.8e308

10 Definire una variabile

Per definire una variabile in C++ si scrive il tipo seguito da uno o più spazi e il nome con alla fine il punt oe virgola. Il nome può essere qualsiasi cosa, ma vengono usate delle convenzioni per creare dei buoni identificaotri.

C++ è case sensitive, il che significa che la variabile `test` e la variabile `Test` sono due variabili diverse.

Naming Convention

Esistono molte convenzioni per nominare una variabile, ad esempio se una variabile è formata da più di una parola allora si possono seguire le seguenti convenzioni: `my_var` o `myVar`, c'è chi fa riferimento alle *Hungarian notation*, l'idea dietro le notazione ungherese è quella di prefissare le variabili con un set di caratteri che ne descrivono il tipo, per esempio una variabile `long` avrà una `l` minuscola avanti, una `int` una `i`.

Determinare la memoria utilizzata da una variabile

Non bisogna dare per scontato lo spazio che occupa una variabile, essendo che questo può variare in base al compilatore o processore, anche se in genere lo spazio occupao difficilmente è diverso dalla tabella sopra citata. Per tanto C++ mette a disposizione un operatore chiamato **sizeof** che se usato su una variabile restituisce il numero di byte che occupa.

```
#include <iostream>

int main() {
    int i;
    std::cout << sizeof(i); // oppure sizeof(int)
}
```

Dichiarare più variabili allo stesso tempo

Si possono dichiarare più variabili sulla stessa linea utilizzando la seguente sintassi:
`int x, y;`

Assegnare un valore ad una variabile

Si può assegnare un valore ad una variabie usando l'operatore d'assegnazione =
`x = 0; // oppure se vogliamo dichiarare e assegnare int x = 0;`

Creare alias con typedef

Quando si lavora con tipi di dati in C++, può diventare noioso, ripetitivo e, soprattutto, soggetto a errori dover scrivere continuamente "unsigned short int". Per semplificare questo processo, C++ consente di creare un alias utilizzando la parola chiave "typedef", che sta per type definition (definizione di tipo).

In pratica, si sta creando un sinonimo per un tipo di dato esistente, e questo è importante distinguere dalla creazione di un nuovo tipo di dato. Per utilizzare "typedef", si scrive la parola chiave, seguita dal tipo di dato esistente e poi dal nuovo nome che si vuole assegnare, terminando con un punto e virgola. Ad esempio:

```
typedef unsigned short int usint; // usint = unsigned short int
```

Con questa dichiarazione, si sta creando un nuovo nome chiamato "usint" che può essere utilizzato ovunque si sarebbe potuto scrivere "unsigned short int".

Costanti

Come le variabili le costanti seervono a memorizzare dati, si distinguono dalle variabili perchè come suggerisce il nome queste non cambiano, rimangono costanti. Bisogna dichiarere e assegnare una quando la si crea e il suo valore non varia nel tempo. C++ ha due tipi di costanti, **letterali** e **simboliche**.

Costanti Letterali

Una costante letterale è una costante rappresentata da un nome, come una variabile, ma a differenza delle variabili il suo valore non cambia.

Ad esempio:

```
age = 20;
```

Costanti Simboliche

Se c'è un caso di questo tipo dove dobbiamo calcolare gli studenti di tutto la scuola:

```
students = classes*15;
```

Una costante letterale è una costante rappresentata da un nome, come una variabile, ma a differenza delle variabili il suo valore non cambia.

In questo esempio sarebbe più comodo se 15 fosse una costante che indica il numero degli studenti per ogni classe.

```
student = classses*StudentPerClas

int age = 39;
```

Esistono due modi in C++ di dichiarare costanti simboliche, la vecchia e ormai obseoleta maniera è con una direttiva del preprocessore con *#define*, la seconda maniera più appropiata è usando la keywork *const*.

Definire costanti con #define

```
#define age 20
```

Definire costanti con const

```
const int age = 20;
```

11 Cos'è un array

Un array p una collezione sequenziale di posizioni di memoria, ognuna delle quali contiene lo stesso tipo di dati. Ogni posizione di memoria è chiamata elemento dell array. Per dichiarare un array si specifica il tipo di dati seguito dal nome dell'array e dalla dimensione tra parantesi qudre.

```
int numeri[6]; // array che contiene due numeri.
```

Quando il compilatore incontra questa dichiarazione preserva abbastanza memoria per contenere tutti gli elementi (in questo caso 6 numeri interi, 4*6 = 24).

Accedere agli elementi dell'array

Gli elemeti di un array vengono referenziati in base al loro **offset** dall'inzio array. Gli offsey degli elementi dell'array sono contati a partire da zero. Quindi il primo elemento dell'array è chiamato `nomeArray[0]`, il secondo `nomeArray[1]` e così via.

```
#include <iostream>

int main() {
    int test[5];
    for(int i = 0; i < 5; i++) {
        std::cin >> test[i];
    }

    for(int i : test) {
    std::cout << i << "\n";
    }
}
```

In questo codice viene inizializzato un array che può contenere 5 elementi, successivamente tramite un ciclo for l'utente inserisce tutti gli elemeneti e poi tramite un altro ciclo for vengono stampati.

12 Array multidimensionali

Un array multidimensionale è un array con più di una dimensione, è una raccolta di elementi in cui ogni elemento viene accesso utilizzando più indici, si dichiara con `datatype[size1][size2]...[sizeN];`. Esempio:

```
int two_d[2][4];
int three_d[2][4][8];
```

Dimensione di un array multidimensionale

La dimensione di un array multidimensionale è uguale alla dimensione del data type moltiplicata per il numero totale di elementi che posso essere contenuti nell'aeeay, questo può essere calcolato moltiplicando la grandezza di ogni dimensione.

```
int arr1[2][4] // grandezza = 2 * 4 * 8 = 32 (byte)
```

Array bidimensionali (2D array)

Un array bidimensionale in C++ è una collezione di elementi organizzati in tighe e colonne, possono essere visualizzati come una tabella/griglia dove ogni elemento è accesso utilizzando due indici: uno per le righe e l'altro per le colonne.

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

Sintassi di un array 2D

```
data_Tyoe array_name[n][m]; // n: numero di righe; m: numero di colonne;
```

Inizzializzazione di array bidimensionali

Ci sono modi differenti di inizializzare un array 2D:

- Utilizzando un elenco di elementi
- Utilizzando i Loop

Utilizzando un elenco di elementi:

```
int main() {
  int x[2][2] = {
    {1, 2},
    {2, 7}
  };

  std::cout << x[1][1];
}
```

Utilizzando i Loop:

```
/*

Crea il seguente array bidimensionale usando cicli for:

{
    {1, 2},
    {2, 4}
}

*/



#include <iostream>


int main() {
    int x[2][2];

    for(int i = 0; i < 2; i++) {
        for(int k = 0; k < 2; k++) {
            if(i == 0 && k == 0) {
                x[i][k] = 1;
            } else if(!(i == 1 && k == 1)){
                x[i][k] = 2;
            } else {
                x[i][k] = 4;
            }
        }
    }
}
```

Per printarlo:

```
for(int z = 0; z < 2; z++) {
    for(int j = 0; j < 2; j++) {
        std::cout << x[z][j] << " ";
    }
    std::cout << "\n";
}
```

Accedere agli elementi degli array bidimensionali

Si possono acedere gli elementi di un array multidimensionale usando gli indici della colonna e della riga.

```
nomearray[i][j]

/*
i -> indice riga
j -> indice colonna

*/
```

Array tridimensionali

todo

String

La libreria standard di C++ include una classe `string` che permette di lavorare faclmente con le stringhe forndendo un set di dati e funzioni per manipolare le stringhe.
`std::string` gestisce i dettagli sull'allocazione della memoria e rende la dichiarazione / assegnazione di stringhe un attività semplice.

```
std::string test = "test";

std::cout << "Contenuto variabile string = " << string << "\n;
```

13 `if` Statement

In un programma C++ il flusso di esecuzione procede normalmente linea per linea nell'ordine in cui appare nel codice sorgente. L'istruzione **if** consente di testare una condizione e di ramificare diverse parti del codice a seconda del risultato.

in C++ il valore 0 viene valutato come false e tutti gli altri come true

```
if(condizione) {
    //fai questo
}
```

L'espressione tra parentesi può esssere qualsiasi espressione, ma di solito contiene una delle espressioni **relazional**. Se la condizione ha il valore `false` allora viene ignorata, se `true` allora viene eseguita.

`else` Statement

Lo Statement `else` consente di eseguire un blocco di codice in caso la condizione if è falsa e nessuna delle condizione else if è vera.

Precedenza logici

Come tutte le espressioni gli operatori logici: true o false, come tutte le espressioni hanno anche un ordine di precedenza ched determina quale relazione viene valutata per prima. Ad esempio:

```
if(x > 5 && y > 5 || z > 5)
```

In questo caso il potrebbe essere che il programmatore voglia che quest'espressione sia valutata come true se sia x che y sono maggiori di 5 o se y è maggiore di 5, oppure se x è maggiore di 5 e o y o z è maggiore di 5. Per sistemare questo problema si possono usare le parentesi.

```
if((x > 5) && (y > 5 || z > 5)) {
    // esegui questo
}
```

Definizioni

Statements In italiano istruzione, controlla la sequenza di esecuzione, valuta un'espressione o non fa nulla. Tutte le istruzioni in C++ terminano con un `;`.

14 Funzioni

Le funzioni si presentano in due varianti: definite dall'utente e incorporate. Le funzioni incorporate fanno parte del tuo pacchetto compilatore — sono fornite dal produttore per il tuo utilizzo. Le funzioni definite dall'utente sono le funzioni che scrivi tu stesso.

Dichiarare e Definire le Funzioni

Valori di Ritorno, Parametri e Argomenti

Come appreso nella Lezione 2, "L'Anatomia di un Programma C++", le funzioni possono ricevere valori e restituire un valore. Quando chiami una funzione, può svolgere un lavoro e restituire un valore come risultato di tale lavoro. Questo è chiamato il suo valore di ritorno, e il tipo di quel valore di ritorno deve essere dichiarato. Quindi, se scrivi

```
int myFunction();
```

stai dichiarando una funzione chiamata myFunction che restituirà un valore intero. Ora considera la seguente dichiarazione:

```
int myFunction(int someValue, float someFloat);
```

Questa dichiarazione indica che myFunction restituirà comunque un intero, ma accetterà anche due valori. Quando invii valori a una funzione, questi valori agiscono come variabili che puoi manipolare dall'interno della funzione. La descrizione dei valori che invii è chiamata lista dei parametri. Nell'esempio precedente, la lista dei parametri contiene someValue, che è una variabile di tipo intero, e someFloat, che è una variabile di tipo float.

Come puoi vedere, un parametro descrive il tipo del valore che verrà passato alla funzione quando la funzione viene chiamata. Gli effettivi valori che passi alla funzione sono chiamati gli argomenti. Considera il seguente esempio:

```
int theValueReturned = myFunction(5, 6.7);
```

Qui vedi che una variabile intera theValueReturned viene inizializzata con il valore restituito da myFunction, e che i valori 5 e 6.7 vengono passati come argomenti. Il tipo degli argomenti deve corrispondere ai tipi dichiarati dei parametri. In questo caso, il 5 va a un intero e il 6.7 va a una variabile float, quindi i valori corrispondono.

Variabili Globali e Locali

Le variabili definite al di fuori di qualsiasi funzione hanno uno "scope" globale e sono quindi disponibili da qualsiasi funzione nel programma, compresa la funzione main(). Le variabili locali con lo stesso nome delle variabili globali non cambiano le variabili globali. Tuttavia, una variabile locale con lo stesso nome di una variabile globale nasconde la variabile globale. Se una funzione ha una variabile con lo stesso nome di una variabile globale, il nome si riferisce alla variabile locale e non a quella globale quando viene utilizzato all'interno della funzione.

Esempio di Utilizzo

```
#include <iostream>

void myFunction();
int x = 5, y = 7;

int main()
{
    using namespace std;

    cout << "x from main: " << x << endl;
    cout << "y from main: " << y << endl << endl;
```

```
    myFunction();

    cout << "Back from myFunction!" << endl << endl;
    cout << "x from main: " << x << endl;
    cout << "y from main: " << y << endl;

    return 0;
}

void myFunction()
{
    using std::cout;
    int y = 10;

    cout << "x from myFunction: " << x << std::endl;
    cout << "y from myFunction: " << y << std::endl << std::endl;
}
```

In questo esempio, le variabili globali x e y sono dichiarate e inizializzate all'inizio del programma. La funzione main() stampa i valori globali di x e y. Quando myFunction() è chiamata, essa definisce una variabile locale y e stampa sia la variabile globale x che la variabile locale y. Al ritorno dalla funzione, la funzione main() stampa nuovamente i valori globali di x e y, dimostrando come le variabili locali non influiscano sulle variabili globali.

Default Parameters

Per ogni parametro dichiarato in un prototipo e in una definizione di funzione, la funzione chiamante deve passare un valore. Il valore passato deve essere del tipo dichiarato. Quindi, se hai una funzione dichiarata come

```
long myFunction(int);
```

la funzione deve, infatti, accettare una variabile di tipo intero. Se la definizione della funzione è diversa o se non si passa un intero, si ottiene un errore del compilatore.

L'unico caso in cui questa regola non si applica è se il prototipo della funzione dichiara un valore predefinito per il parametro. Un valore predefinito è un valore da utilizzare se nessuno viene fornito. La dichiarazione precedente potrebbe essere riscritta come

```
long myFunction(int x = 50);
```

Questo prototipo dice: "myFunction() restituisce un long e accetta un parametro intero. Se non viene fornito un argomento, utilizza il valore predefinito di 50". Poiché i nomi dei parametri non sono richiesti nei prototipi delle funzioni, questa dichiarazione potrebbe essere stata scritta anche come

```
long myFunction(int = 50);
```

La definizione della funzione non cambia dichiarando un parametro predefinito. L'intestazione della definizione di funzione per questa funzione sarebbe

```
long myFunction(int x)
```

Se la funzione chiamante non include un parametro, il compilatore riempirà x con il valore predefinito di 50. Il nome del parametro predefinito nel prototipo non deve essere lo stesso del nome nell'intestazione della funzione; il valore predefinito è assegnato per posizione, non per nome.

È possibile assegnare valori predefiniti a tutti o solo ad alcuni dei parametri della funzione. L'unica restrizione è la seguente: se un parametro non ha un valore predefinito, nessun parametro precedente può avere un valore predefinito.

Se il prototipo della funzione ha questa forma

```
long myFunction(int Param1, int Param2, int Param3);
```

puoi assegnare un valore predefinito solo a Param2 se hai assegnato un valore predefinito anche a Param3. Puoi assegnare un valore predefinito a Param1 solo se hai assegnato valori predefiniti sia a Param2 che a Param3. La Lista 6.7 illustra l'uso dei valori predefiniti.

Overloading Function

In C++, è possibile creare più di una funzione con lo stesso nome, caratteristica nota come *sovraccarico di funzioni* (*function overloading*). Le funzioni sovraccaricate devono differire nella loro lista di parametri con un tipo di parametro diverso, un numero diverso di parametri o entrambi. Un esempio è il seguente:

```
int myFunction(int, int);
int myFunction(long, long);
int myFunction(long);
```

La funzione `myFunction()` è sovraccaricata con tre elenchi di parametri. Le prime due versioni differiscono nei tipi dei parametri, mentre la terza differisce nel numero di parametri. I tipi di ritorno possono essere gli stessi o diversi, ma le versioni sovraccaricate non possono differire solo nel tipo di ritorno. È importante che una versione sovraccaricata di una funzione presenti una firma unica in termini di tipo di argomenti accettati.

Le funzioni con **lo stesso nome e lista di parametri, ma con tipi di ritorno diversi, generano un errore del compilatore**. Il sovraccarico di funzioni è anche chiamato *polimorfismo di funzioni*, che significa la capacità di sovraccaricare una funzione con più di un significato. Cambiando il numero o il tipo dei parametri, è possibile dare a due o più funzioni lo stesso nome, e la giusta funzione sarà chiamata automaticamente corrispondendo ai parametri utilizzati.

Il "sovraccarico" di funzioni consente di creare una funzione in grado di calcolare la media di interi, double e altri valori senza dover creare nomi individuali per ogni funzione. Ad esempio:

```
int Double(int);
long Double(long);
float Double(float);
double Double(double);
```

Questo approccio è più leggibile e più semplice da utilizzare rispetto alla creazione di nomi di funzioni separati per ogni tipo di dato.

Function - Recursion

TODO

15 Loop - Control Program Flow

Molti problemi sono risolti lavorando rapidamente sullo stesso dato, Ci sono due modi per dare ciò, la *recursione* e l'**iterazione**, il metodo principale per iterare sono i **loop**.

Il principio dei loop: `goto`

Agli albori dell'informatica i programmi erano brevi e "fatti male", i loop consistevano in un'etichetta (label) alcune iarazioni e istruzione, un salto che andava all'etichetta. In c++ un etichetta (label) non è altro che un nome seguito da i due punti. Un salto (jump) è eseuito scrivendo `goto` seguito dal nome del label.

Example 15.1.

```
#include <iostream>

int main() {
   int i = 0;

    nomeetichetta:
       if(i<10){
           std::cout << i << "\n";
           i++
           goto nomeetichetta; // ritorna a nomeetichetta
       }
    std::cout << "fine";
}
```

`while` Loops

Un loop while fa in modo che il programma ripete una sequenza di statements finchè una determinata condizione rimane vera.

Example 15.2.
```
while{i < 5}{
i++;
std::cout << i;
}
```

Questo codice incerementa i di 1 e lo stampa finchè questo è minore di 5.

`continue` e `break`

Lo statements continue torna all'inizio del loop, mentre lo statements break interrompe il loop.

`while(true)`

Un ciclo while si ripete finchè la condizione tra partentesì è verificata quindi possiamo creare un loop infinito con una condizione sempre verificata.

`do while` Loop

Quando viene creato un ciclo while c'è la possibilità che questo non venga mai eseguito, se la condizione non è verificata l'intero corpo del'loop verrà saltato.

Il loop `do while` esegue prima il corpo del'loop e poi verifica la condizione, quindi esegue il corpo del'loop almeno una volta.

Example 15.3.
```
do {
i--;
std::cout << "ciao"

} while(i>0)
```

Si usa il loop do while quando si vuole che il corpo del loop venga eseguito almeno una volta

`for` Loop